SKILLFACTORY: SELF-DISTILLATION FOR LEARNING COGNITIVE BEHAVIORS

Anonymous authors
Paper under double-blind review

ABSTRACT

Reasoning models leveraging long chains of thought employ various cognitive skills such as verification of their answers, backtracking, retrying by an alternate method, and more. Previous work has shown that when base models exhibit these skills, a reasoning model trained by reinforcement learning (RL) can learn to leverage them. How can we get models to leverage skills that aren't exhibited by base models? Our work, SkillFactory, is a method for fine-tuning models to roughly learn these skills during a supervised fine-tuning (SFT) stage prior to RL. Our approach does not rely on distillation from a stronger model, but instead uses samples from the model itself, rearranged to provide training data in the format of those skills. These "silver" SFT traces may contain errors, but are nevertheless effective for priming a model to acquire skills during RL. Our evaluation shows that (1) starting from SkillFactory initialization helps a model post-RL to generalize to harder variants of the task; (2) cognitive skills are indeed used by the model; (3) the presence of these skills allows for opportunities like budget forcing (driving a model to think longer) that other baselines lack.

1 INTRODUCTION

Modern large language models (LLMs) increasingly demonstrate the ability to acquire and apply a variety of cognitive behaviors we can call "skills." These include capabilities such as systematically exploring a solution space, verifying outputs, and retrying with alternative strategies (Marjanović et al., 2025). Such skills are particularly valuable for reasoning, as they enable models to explore different paths to a solution rather than relying on a single attempt (Bogdan et al., 2025). Indeed, many of the major gains in reasoning-focused LLMs over the past year can be traced to better elicitation of these skills during inference time, demonstrating that skill acquisition itself has become a primary driver of progress in reasoning (Jaech et al., 2024; Guo et al., 2025; Abdin et al., 2025).

Reinforcement learning (RL) has proven to be a powerful paradigm for unlocking many of these capabilities (Guo et al., 2025). If a model already demonstrates these skills, or is equipped with them through distillation or continued pre-training, then RL can further reinforce these behaviors (Gandhi et al., 2025). However, this often requires access to superior models (Muennighoff et al., 2025; Guha et al., 2025), significant training (Yeo et al., 2025), custom pretraining data, or a complex mix of all of these. These methods have limitations on the flexibility and generalization of the instilled skills.

In this work, we propose **SkillFactory**, a framework to instill these behaviors into models and unlock large gains from RL *without* distilling from a larger model. Through prompting and restructuring of the samples into a structured output, we can construct "silver" traces (which may contain errors) that demonstrate a model verifying its outputs and retrying based on failures; see Figure 1. A model trained on this data with supervised fine-tuning (SFT) is not yet calibrated to use these skills effectively; however, past work suggests that learning the form of skills may be enough Li et al. (2025) and the model may be primed for effective RL. The RL stage hones the skills instilled into the model, improving both how they are used and where. Furthermore, the explicit nature of how our skills are invoked allows for advanced decoding methods at inference time, like budget forcing.

Contributions We demonstrate that (1) models can acquire complex reasoning skills from their own rearranged outputs without requiring stronger teacher models; (2) SkillFactory initialization enables generalization to harder task variants and novel domains post-RL, matching or exceeding

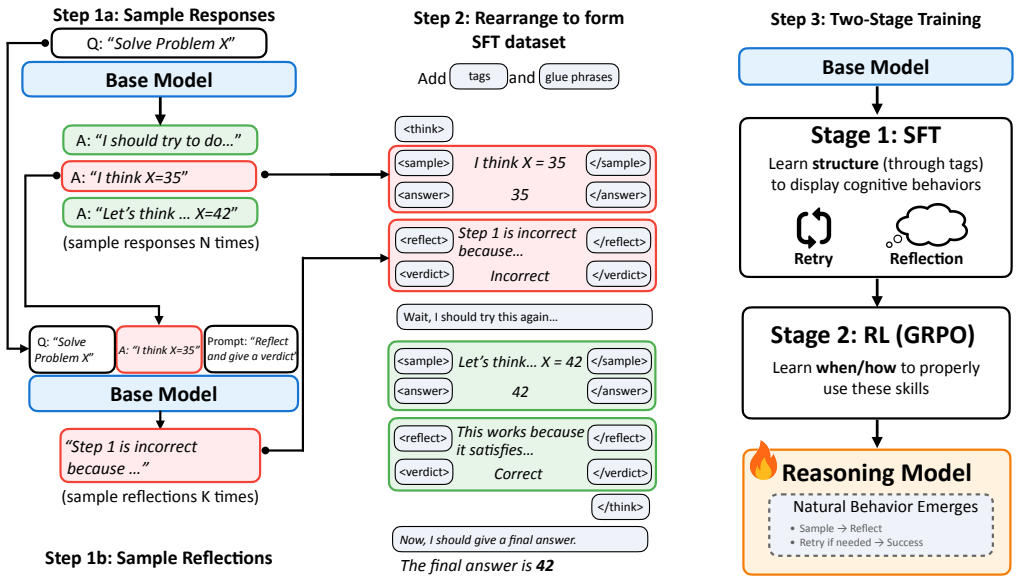

Figure 1: SkillFactory framework. We obtain responses and reflection traces using a model's own sampled reasoning, then rearrange them to demonstrate reasoning skills. A model SFTed on this data is an effective starting point for RL, yielding better performance and more skill usage post-RL.

the performance of strong baselines; and (3) our approach enables advanced inference techniques like budget forcing that baseline methods cannot support.

2 Background and Motivation

2.1 Cognitive Skills in LLMs

LLMs take in an input $\mathbf{x}$ and place a distribution $p(\mathbf{y} \mid \mathbf{x})$. For the tasks we consider, we assume a final answer can be extracted via a process $a = \texttt{extract}(\mathbf{y})$ (e.g., if it is embedded in <answer> tags). Large *reasoning* models fit in this framework, but are characterized by two differences: (1) they exhibit the use of reasoning skills rather than simple "linear" solving processes; (2) their outputs $\mathbf{y}$ are typically much longer. Past work describes a number of cognitive skills useful for reasoning (Gandhi et al., 2025). In this work, we focus on the following two:

1. **Retrying:** A prefix $\mathbf{y}_{<i}$, where i is length in tokens, ends in an answer $\tilde{a} = \texttt{extract}(\mathbf{y}_{<i})$. The model decides to restart its inference, generating tokens like "*Wait, let me rethink this...*" and generating completion $\mathbf{y}_{\geq i}$ with little apparent connection to what came before.
2. **Reflection:** A prefix $\mathbf{y}_{<i}$ ends in an answer $\tilde{a} = \texttt{extract}(\mathbf{y}_{<i})$. The model enters a separate process of verifying $\tilde{a}$, generating tokens $\mathbf{y}_{v(\tilde{a})}$ focused on evaluating the answer.

Together, these methods guide the model to generate long chain-of-thoughts beyond a single attempt, leading to more robust reasoning.

Existing usage of skills and the need for SkillFactory A central finding of Gandhi et al. (2025) is that some base language models already exhibit these skills in some form. Figure 2 shows an example of this for the Countdown number puzzle, where the task is to combine a set of input numbers using the four basic arithmetic operations $(+, -, \times, \div)$ to reach a target number. A red highlight shows the model verifying the outcome of the computation as incorrect; highlighted in blue is an instance where the model restarts and tries to find another solution.

Two fundamental observations underlie our work. First, **these skills surface less consistently when incidentally expressed in natural language**. We will see in our results that SkillFactory consistently leads to longer traces exhibiting phenomena like verification and retries to a higher extent than the base model, particularly on out-of-domain tasks.

Second, **there are advantages to explicitly marking skills' usage**. Our format, shown in Figure 3, has reflection *explicitly demarcated* by a tag. The reflection process also reruns the computation, potentially mitigating errors that happened during the initial search. Retrying is similarly marked by a tag. These not only lead to more effective usage of the skills, but also allow us to apply budget forcing (Muennighoff et al., 2025) in a principled way.

Finally, although we do not focus on demonstrating it in this work, SkillFactory allows us to impart reasoning skills that are unattested in the base model. We therefore view this work as providing a platform for shaping cognitive behaviors of LLMs across a variety of tasks.

[…]
Let's try a different combination:

1. First, add 98 and 20:
   Step 1: 98 + 20 = 118
2. Then, add 44 to the result:
   Step 2: 118 + 44 = 162

This is also not equal to 102. Let's try a simpler approach with a different order:

1. First, subtract 20 from 98:
Step 1: 98 - 20 = 78
2. Then, add 44 to the result:
Step 2: 78 + 44 = 122
[…]

Figure 2: Trace from Countdown exhibiting implicit reflection and retrying.

Existing Approaches to Eliciting Reasoning Skills Current methods for developing reasoning capabilities in language models can be broadly categorized into three main approaches. First, RL with sparse rewards can surface reasoning behaviors latent in the base model (Shao et al., 2024; Yu et al., 2025; Liu et al., 2025). This approach relies heavily on a strong base model, and these skills may fail to emerge naturally when not sufficiently represented in the pretraining data; our results show that pure RL does not yield robust skill use in cross-task generalization. Second, distillation from stronger models (Muennighoff et al., 2025; Ye et al., 2025; Guha et al., 2025), provides training data exhibiting advanced reasoning, though this requires access to superior models and often struggles to generalize beyond the domains of the distilled data (Gudibande et al., 2024; Kalai et al., 2025). Third, targeted data curation, through continual pretraining on backtracking examples (Gandhi et al., 2025), hand-crafted reasoning chains for in-context learning (Pang et al., 2025), or Monte Carlo tree search rollouts (Kim et al., 2025, ASTRO), has shown promise in instilling specific cognitive skills before or during fine-tuning. SkillFactory is similar to these methods, but focuses on generating data entirely from the base model and highlights that structure is key for the generalization of consistent skill use.

Additional Related Work Reasoning models face two failure modes: overthinking (excessive verbosity) (Sui et al., 2025) and underthinking (prematurely abandoning solution paths) (Wang et al., 2025). While some argue for fundamental limitations in reasoning models (Shojaee et al., 2025) due to problem complexity. We argue that skills offer a way to explore solution paths and exit them effectively when learned correctly. This allows for heuristic reasoning methods to be instilled into the model, expanding what they can solve.

2.2 Tasks: Planning, Search, and Computation

The usefulness of cognitive skills varies across tasks. While a skill like verification can in principle be used anywhere, it is more effective on "NP-complete"-like tasks: those that are easier to check than to generate answers for. LLMs also vary in how much they naturally exhibit these skills.

We define three types of tasks capturing the range of relevant phenomena for our approach: **search-focused**, **computation-focused**, and **other**. A full set of tasks can be found in Section 4.2.

Search-focused tasks are those like Countdown (Figure 2). The space of possible responses is usually large, and an LLM is expected to execute search in its context to find an answer. Verification and retrying are *naturally exhibited* by models, although inconsistently, and verification is highly effective, since the solutions are easier to check than they are to find.

Computation-focused tasks include multiplication and GSM8K (Cobbe et al., 2021). These tasks are ones where LLMs on the scale we experiment with are prone to making mistakes. However, verification and retrying are *not naturally exhibited*.

Finally, **other** tasks include CommonsenseQA (Talmor et al., 2019). We use CommonsenseQA to help spot if the model has degraded performance on other tasks from training on other domains.

3 SKILLFACTORY

SkillFactory has three pieces, depicted in Figure 1. (1) Data curation: uses inference on a base model in combination with heuristics tied to each cognitive skill of interest. (2) Supervised fine-tuning: the model is fine-tuned on these traces. Unlike other distillation approaches, we don't expect performance to increase in this step; we are only trying to achieve a better starting point for RL. (3) Reinforcement learning: We use off-the-shelf RL algorithms such as GRPO (Shao et al., 2024; Marjanović et al., 2025), combined with sparse rewards based on correctness. We focus on the data curation stage in this section.

We generate SkillFactory data in three steps: sampling diverse solutions from the base model, generating reflections that assess those solutions, and combining them into structured traces that exhibit explicit retry and verification behaviors. Throughout this process, we use $\mathbf{y}$ to denote solution attempts and $\mathbf{r}$ to denote reflections. Algorithm 1 outlines the complete procedure, and each of the three aforementioned steps will be detailed in the succeeding paragraphs.

Algorithm 1 SkillFactory Trace Construction. All values of the parameters used in the Trace Construction algorithm can be found in Table 7 of the Appendix.

Require: Dataset $D_T = \{(\mathbf{q}_i, \mathbf{a}_i)\}$, base model $\mathcal{M}$, prompts $P_{\text{solve}}, P_{\text{reflect}}$
Ensure: Training set $\mathcal{D}_{\text{SFT}}$

$\mathcal{D}_{\text{SFT}} \leftarrow \emptyset$
for each question $(\mathbf{q}_i, \mathbf{a}_i) \in D_T$ **do**
  // Generate solution-reflection pairs
  Sample solutions: $\mathcal{Y} \leftarrow \{\mathbf{y}_j \sim \mathcal{M}(\mathbf{q}_i \mid \mathbf{p}) : \mathbf{p} \in P_{\text{solve}}, j \in \{1, 2, \ldots, N_{\text{sample}}\}$
  Generate reflections: $\mathcal{R} \leftarrow \{\mathbf{r} \sim \mathcal{M}(\mathbf{q}_i, \mathbf{y} \mid P_{\text{reflect}}) : \mathbf{y} \in \mathcal{Y}, \text{verdict}(\mathbf{r}) = \text{correct}(\mathbf{y}, \mathbf{a}_i)\}$
  $\mathcal{Y}^+ \leftarrow \{(\mathbf{y}, \mathbf{r}) : \texttt{correct}(\mathbf{y}, \mathbf{a}_i) = \text{True}\}$ ▷ correct pairs
  $\mathcal{Y}^- \leftarrow \{(\mathbf{y}, \mathbf{r}) : \texttt{correct}(\mathbf{y}, \mathbf{a}_i) \neq \text{True}\}$ ▷ incorrect pairs
  while $|\mathcal{Y}^+| > 0$ **do**
    // Determine trace length
    $n^+ \leftarrow \min(\text{Uniform}([1, L_{\text{max}}]), |\mathcal{Y}^+|)$
    $n^- \leftarrow \min(\text{Uniform}([0, n^+ - 1]), |\mathcal{Y}^-|)$
    // Sample solution-reflection pairs
    $T^+ \leftarrow$ sample n^+ items from $\mathcal{Y}^+$ without replacement
    $T^- \leftarrow$ sample n^- items from $\mathcal{Y}^-$ without replacement
    // Build trace, ensuring that it ends on a correct solution
    $\mathbf{trace} \leftarrow \text{shuffle}(T^- \cup T^+[1 : n^+ - 1]) \cup \{T^+[n^+]\}$ ▷ Append last correct
    // Format into training instance
    $\mathcal{D}_{\text{SFT}} \leftarrow \mathcal{D}_{\text{SFT}} \cup \{\texttt{format}(\mathbf{q}_i, \mathbf{trace})\}$
return $\mathcal{D}_{\text{SFT}}$

Solution Generation For each question $\mathbf{q}_i$ in our task dataset $D_T = \{(\mathbf{q}_i, \mathbf{a}_i)\}_{i=1}^n$, we sample N_{sample} solution attempts from our base model $\mathcal{M}$. To encourage diversity, we use a set of four different chain-of-thought prompts P_{solve}. For each prompt, we sample 16 responses, yielding a solution set $\mathcal{Y}$ of 64 attempts per question. The full set of prompts can be found in Appendix C.2.

Each solution $\mathbf{y} \in \mathcal{Y}$ is automatically verified: we use $\texttt{extract}(\mathbf{y})$ to parse the final answer from the solution and check if it matches ground truth $\mathbf{a}_i$. Since SkillFactory prompts the model to enclose its final answer in `<answer>` tags, our `extract()` function leverages these tags for parsing. We define $\texttt{correct}(\mathbf{y}, \mathbf{a}_i) = \mathbb{1}[\texttt{extract}(\mathbf{y}) = \mathbf{a}_i]$ to indicate whether a solution is correct. This gives us a pool of both correct and incorrect solutions, both are needed to teach the model self-correction.

Reflection Generation Next, we prompt $\mathcal{M}$ to reflect on each solution attempt using a reflection prompt p_{reflect}. A reflection $\mathbf{r}$ critiques the reasoning in solution $\mathbf{y}$ and predicts its correctness, $\texttt{correct}(\mathbf{y}, a_i)$. We use $\texttt{verdict}(\mathbf{r})$ to extract this prediction from the reflection text. Just like with the answer tags, SkillFactory also prompts the model to use `<verdict>...</verdict>` tags when generating reflections, which we then use for parsing the verdicts. A valid reflection is one where $\texttt{verdict}(\mathbf{r}) = \texttt{correct}(\mathbf{y}, \mathbf{a}_i)$. The reflection prompts can be found in Appendix C.3.

We sample four reflections per solution but keep only those where $\text{verdict}(\mathbf{r}) = \text{correct}(\mathbf{y}, \mathbf{a}_i)$, reflections that accurately judge whether the solution succeeded or failed. The result is a set $\mathcal{R}$ of valid reflections paired with their corresponding solutions.

Trace Construction Finally, we assemble solution-reflection pairs into training traces. We partition our pairs into correct ($\mathcal{Y}^+$) and incorrect ($\mathcal{Y}^-$). For each trace, we:

- Sample n^+ correct pairs and n^- incorrect pairs, where $n^- < n^+$
- Shuffle all but one correct pair to create a mixed sequence
- Append the remaining correct pair to ensure success at the end
- Format the sequence using `format()`, which wraps each solution-reflection pair in tags and adds transition phrases; see Figure 3.

This creates traces where the model attempts a problem, reflects on its work, tries again if needed and always eventually succeeds. The `format()` function applies the template shown in Figure 3, interleaving solutions with reflections in `<sample>` and `<reflect>` tags respectively. Pairs of samples and their reflections are concatenated together with phrases like "*Let me reconsider*". By training on these restructured outputs, we prime the model to employ these skills during RL. A full list of phrases used to stitch together the pairs can be found in Appendix C.1.

```
User: [question]
Assistant: <think>
[Attempt 1]
Reflect: "Wrong because..."
Let me try again.
[Attempt 2]
Reflect: "Need to verify..."
...
[Final correct attempt]
[Reflection: "This looks correct..."]
</think>
Answer: [final answer]
```

Figure 3: SkillFactory training trace with self-reflection and retry.

4 Experimental Setup

4.1 Baselines

We evaluate SkillFactory against four baselines, each representing a different paradigm for developing reasoning models as outlined in Section 2.

RL Only Our first baseline trains directly from the base model Qwen2.5-1.5B-Instruct (Team, 2024) using only reinforcement learning with binary correctness rewards. This represents the simplest approach by relying on RL to discover reasoning behaviors. We use the same GRPO setup as SkillFactory, but start from the unmodified base model.

BOLT (external data curation) We establish a baseline similar to the flow of BOLT (Pang et al., 2025) where we (1) Sample 10 in-context learning examples from a strong reasoning model (Claude Sonnet 4), (2) prompt an LLM (GPT-4o-mini) with ICL to generate reasoning traces for new problems, creating synthetic SFT data, and (3) train the resulting model using GRPO. We provide additional details in Appendix E.

Distillation (learning from strong models) We also evaluate conventional distillation (Muennighoff et al., 2025; Ye et al., 2025; Guha et al., 2025), where we train on traces from a more capable model. We prompt R1 to solve problems from our training set and collect its generated reasoning traces. We perform SFT only on these traces (no RL stage).

STaR (learning from correct outputs) Finally, we compare with a self-distillation method called STaR (Zelikman et al., 2022). STaR iteratively samples from the base model, checks if the answer is correct, and subsequently uses it to train the model if the answer is correct. We perform this for our base model then train with RL.

These baselines allow us to isolate the contribution of SkillFactory's self-distillation with structured rearrangement against: learning from scratch (RL Only), sophisticated external data generation (BOLT), traditional distillation (R1), and other commonly used self-distillation methods (STaR).

Notably, SkillFactory and STaR are the only approaches that require neither stronger models nor external data generation, relying entirely on the base model's own outputs.

4.2 Task Setup and Evaluation

We train all methods on **Countdown with 3 arguments**. Countdown requires the model to take a set of input numbers and apply mathematical operations $+, -, \times, \div$ to reach a target. The inputs can be used in any order, but each number can be used at most once. For example, a valid solution for the input numbers $\{17, 11, 8\}$ with target 105 might be $11 \times 8 + 17$ (Gandhi et al., 2024).

Countdown is a task that can encourage the emergence of skills like retrying and reflection naturally because it naturally requires search to solve the question. We evaluate on additional tasks to assess the model's reasoning and determine if the skills generalize.

Harder variants of training tasks:

- *Countdown with 4, 5, and 6 arguments*: This is the same setting as the 3-arg training task. These variants adjust the number of arguments, which can exponentially increase the search space.

Out-of-distribution reasoning tasks:

- *Multiplication* (Dziri et al., 2023): Multiply two N digit numbers. We evaluate $N = 2, 3, 4, 5$.
- *Letter Countdown (CD)*: Given a jumbled set of N letters, rearrange them to form a valid word. We evaluate $N = 4, 5$.
- *Acronym generation*: Given a sequence of N words, create an acronym from the first letters of the words. We evaluate $N = 4, 5$.
- *CommonsenseQA (CSQA)* (Talmor et al., 2019): Multiple choice reasoning questions about commonsense situations.
- *GSM8K* (Cobbe et al., 2021): Grade-school math word problems

4.3 Training

We fine-tune Qwen2.5-1.5B-Instruct (Team, 2024) using silver traces generated from 4,000 Countdown 3arg instances, followed by GRPO reinforcement learning on 1,000 held-out questions for 50 epochs. Full hyperparameters are provided in Appendix A.1.

5 Results

We evaluate SkillFactory across three dimensions: (1) performance on the training task and its harder variants, (2) generalization to out-of-distribution tasks, and (3) the effectiveness of budget forcing at inference time. All results are averaged over four runs with the evaluation settings in Appendix A.3.

5.1 In-Domain Performance

Table 1 shows performance on Countdown tasks of varying difficulty. All methods achieve near-perfect accuracy (98-100%) on the training task (Countdown 3-arg), demonstrating that sparse rewards alone suffice for in-distribution problems.

The key differences emerge on harder variants. On Countdown 4-arg, SkillFactory achieves an accuracy of 30.7%, which is 12.9% higher than the leading baseline (RL-Only at 17.5%). This advantage diminishes as task difficulty increases: on 5-arg and 6-arg, SkillFactory remains competitive but converges toward baseline performance as all methods approach their limits. These results demonstrate that the structured reasoning skills learned through SkillFactory can transfer to moderately harder instances of the same task family, but with diminishing returns on more challenging variants.

5.2 Generalization to Harder Tasks

Table 2 shows the performance of methods trained on Countdown 3-arg when applied to other tasks. While most baselines incur performance degradation after specialized training on Countdown, Skill-

Table 1: Performance on Countdown tasks for models trained on Countdown 3-arg. SkillFactory achieves substantial gains on 4-arg, but skill effectiveness drops as task difficulty increases.

Model	**Countdown**			
	3arg	4arg	5arg	6arg
Qwen2.5-1.5B-Instruct	14.3 ± 2.1	6.0 ± 1.5	2.0 ± 0.8	0.8 ± 0.5
SkillFactory SFT	17.1 ± 1.3	5.2 ± 0.8	2.0 ± 0.5	0.9 ± 0.3
RL Only	99.2 ± 0.5	17.5 ± 2.3	12.7 ± 1.8	**11.2** ± 1.6
R1 Distill	58.4 ± 2.1	15.1 ± 1.4	6.3 ± 1.1	6.4 ± 1.0
BOLT	**99.3** ± 0.4	16.1 ± 2.2	10.4 ± 1.7	9.3 ± 1.5
STaR	99.0 ± 0.5	11.2 ± 1.9	7.6 ± 1.4	7.3 ± 1.3
SkillFactory	98.8 ± 0.5	**30.7** ± 2.2	**12.8** ± 1.6	9.0 ± 1.3

Table 2: Performance on OOD tasks for models trained on Countdown 3-arg. Many models overfit and degrade performance, but SkillFactory achieves positive transfer to other tasks in aggregate.

Model	**Acronym**		**CSQA**	**GSM8k**	**Letter CD**		**Multiplication**				**Overall**
	4	5			4	5	2dig	3dig	4dig	5dig	
Qwen2.5-1.5B-Instruct	11.2	**16.7**	55.6	58.8	15.7	7.0	76.8	**39.8**	5.2	**0.7**	28.8
SkillFactory SFT	3.9	2.3	47.8	60.0	12.5	4.8	86.4	36.4	5.2	0.6	26.0
RL Only	12.3	8.2	**62.8**	68.3	18.2	5.9	81.8	14.5	1.7	0.1	27.4
R1 Distill	10.0	9.4	56.8	62.9	10.7	4.0	79.8	38.7	**7.1**	0.6	28.0
BOLT	**14.5**	10.1	61.6	**70.1**	18.3	6.7	83.7	18.8	2.1	0.3	28.6
STaR	11.9	9.2	60.8	68.3	13.6	5.0	81.6	11.3	0.8	0.1	26.3
SkillFactory	11.8	9.7	60.9	67.7	**20.2**	**9.0**	**94.0**	39.3	6.8	0.7	**32.0**

Factory maintains or improves performance on 9 out of 10 OOD tasks, achieving the highest overall accuracy (32.0%) across all tasks. The imparted skills improve the ability of the model to transfer to new domains. This is further explored in Section 6.

On Letter CD, the most closely related task, SkillFactory yields improvement on both Letter CD 4 and 5, whereas no baseline method improves on Letter CD 5. On Multiplication, a more distantly-related task involving computation, SkillFactory and R1 Distillation maintain their performance, 39.3% and 38.7% respectively, on 3-digit vs. the base model's 39.8%. All other baselines suffer performance degradation, including RL Only (dropping to 14.5%). Additionally we see that the SFT model gains performance on Countdown but begins to degrade performance on OOD tasks after a single epoch of training.

5.3 Budget Forcing

Because SkillFactory teaches explicit retry patterns with structured tags (`<sample>`, `<reflect>`), we can causally intervene at inference time to force additional reasoning, which is an attractive method of test-time scaling. "Budget forcing", explored in s1 (Muennighoff et al., 2025) uses manual editing of traces with linguistic indicators; we instead explore appending a `<sample>` tag before the final answer to trigger another reasoning attempt.

We compare our approach to RL-Only and R1 Distill. They lack structure, so we use the phrases "Let's try a different approach." and "Another thought:" respectively, encouraging the model to think more before giving an `<answer>` tag. Both phrases occur naturally in these models generations.

Table 3 shows results from applying this intervention to every example in the test set across our tasks. SkillFactory shows consistent improvements across Countdown variants ($+8.0\%$ on 4-arg, $+3.6\%$ on 5-arg, and $+4.2\%$ on 6-arg). Multiplication 3-digit sees substantial gains across all methods, with RL Only achieving $+16.7\%$ (though from a lower baseline).

We see two contributing factors to these gains. First, the model **repairs degeneration**. Particularly on harder Countdown variants, models often get stuck in unproductive search patterns and never terminate. Budget forcing enables a model to explore alternative paths to reach an answer. Second, the model exhibits **extended verification**. On computational tasks like Multiplication, the intervention prompts models to reconsider their answers, catching errors.

Table 3: Performance breakdown on out-of-distribution tasks (tasks as rows, methods as column groups). "Std" indicates results prior to budget forcing, and "BF" indicates results with the budget-forcing technique appropriate for that model.

Task	**RL Only**			**R1 Distill**			**SkillFactory**		
	Std	BF	Δ	Std	BF	Δ	Std	BF	Δ
Countdown 4arg	17.5	17.9	0.4	11.6	19.3	7.6	30.7	38.7	**8.0**
Countdown 5arg	12.7	14.0	1.3	5.0	8.1	3.1	12.8	16.4	**3.6**
Countdown 6arg	11.2	13.1	1.9	4.7	8.1	3.4	9.0	13.2	**4.2**
Acronym 4	12.3	9.5	-2.8	9.4	11.9	**2.5**	11.8	12.1	0.3
Acronym 5	8.2	6.4	-1.7	8.9	10.1	**1.2**	9.7	9.0	-0.7
CSQA	62.8	62.8	0.1	50.9	52.1	**1.3**	60.9	59.8	-1.0
GSM8k	68.3	68.8	**0.5**	51.3	49.6	-1.7	67.7	66.1	-1.6
Letter CD 4	18.2	17.0	-1.2	10.0	11.5	**1.5**	20.2	16.5	-3.8
Letter CD 5	5.9	6.8	**0.8**	3.9	4.1	0.2	9.0	7.8	-1.3
Multiplication 2dig	81.8	90.4	**8.5**	66.0	63.4	-2.6	94.0	93.2	-0.8
Multiplication 3dig	14.5	31.2	**16.7**	30.9	30.4	-0.6	39.3	45.2	5.9
Multiplication 4dig	1.7	3.9	**2.2**	5.8	6.0	0.1	6.8	7.0	0.2
Multiplication 5dig	0.1	0.2	0.0	0.5	0.5	0.0	0.7	0.8	**0.0**
Overall	24.2	26.3	2.1	19.9	21.2	1.2	28.7	29.7	1.0

SkillFactory benefits from both mechanisms due to its explicit training on retry and verification patterns. We observe that RL Only primarily gains from the second mechanism but struggles to effectively restart reasoning, while R1 Distillation shows behavior somewhere in between. Importantly, while RL Only shows larger relative improvements on some tasks, SkillFactory maintains superior absolute performance both before and after budget forcing.

6 Ablations and Analysis

6.1 Analysis of Skill Usage

Skill Usage By making reflection steps explicit, we enable evaluation of this step for effectiveness.

Table 4: Number of explicit answer attempts, explicit reflections and the verification F1 for the correct and incorrect classes for Skill Factory.

	#Answers	#Reflect	F1 (correct / incorrect)
Countdown 3arg	1.57	1.22	0.96 / 0.91
Countdown 4arg	1.21	3.06	0.68 / 0.95
Letter CD 4	2.12	1.65	0.34 / 0.79
Multiplication 3dig	2.15	1.84	0.36 / 0.80

Table 4 shows the average number of explicit answer attempts (final answers given in answer tags), average number of explicit reflections (explicit reflection and verification done in reflection tags) and the verification F1 for SkillFactory across both correct and incorrect classes (how often the verifier in the reflection tags makes the ground-truth correct decision about a proposed answer). Reflection as a skill generalizes to other domains, and scales with task difficulty: Countdown 4-arg exhibits more reflection than Countdown 3-arg.

Furthermore, reflection is highly effective: the "incorrect" class F1 indicates that the verifier is highly effective at correctly rejecting wrong answers. The weaknesses of the model itself impair its function; for instance, on Letter Countdown, we see the model exhibiting uncertainty about what is and isn't an English word, suggesting a limitation of our model scale. We report skill use results for all tasks in Appendix D.1.

Length Figure 4 shows that SkillFactory consistently produces responses that are intermediate in length for in-domain tasks (countdown 4-arg) as well as out-of-domain tasks (like multiplication and letter countdown). SkillFactory enables the model to use retry and reflection as skills, preventing it from immediately answering the question always or getting stuck on the problem. On the other hand, the RL baseline tends to only give intermediate outputs for the in-domain variant of the task. For the out-of-domain tasks, the baseline either directly answers the question or just keeps generating. In Appendix D we have sample traces from the RL baseline model and SkillFactory. We qualitatively see evidence that SkillFactory has both *implicit* and *explicit* skill use for countdown variants. For

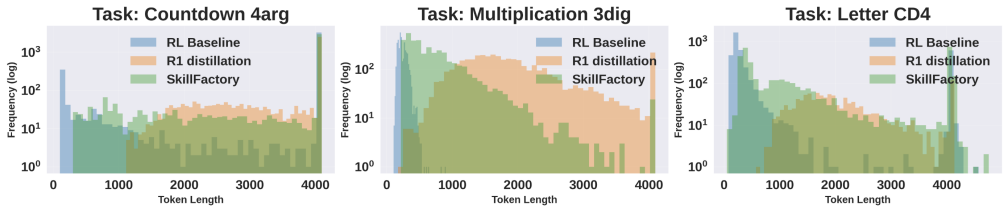

Figure 4: Token length distribution for three tasks for responses given by (a) RL Baseline, (b) R1 distillation, (c) SkillFactory.

Table 5: OOD task results, showing ablations underperform SkillFactory in aggregate.

Model	Acronym		CSQA	GSM8k	Letter CD		Multiplication				Overall
	4	5			4	5	2dig	3dig	4dig	5dig	
Qwen2.5 1.5B Instruct	11.2	**16.7**	55.6	58.8	15.7	7.0	76.8	**39.8**	5.2	**0.7**	28.8
SkillFactory	**11.8**	9.7	60.9	67.7	20.2	**9.0**	**94.0**	39.3	**6.8**	0.7	**32.0**
Instruction Prompt	7.9	6.4	54.9	59.9	12.4	5.2	81.9	28.5	1.1	0.2	25.8
No Sample Order	8.0	5.9	59.3	67.0	10.5	5.2	69.1	14.9	0.6	0.1	24.1
No Reflections	7.4	6.8	57.7	61.5	9.3	4.8	70.2	14.0	0.7	0.2	23.3
No Prompt Diversity	8.4	4.3	**62.4**	**68.5**	**20.3**	7.8	85.8	30.2	2.0	0.3	29.0

out-of-domain tasks, our model still maintains the use of and *explicit* skills. Compared to this, the RL baseline model only transfers skill use to in-domain tasks.

6.2 Ablations

We conduct ablations to understand which components of SkillFactory contribute to its effectiveness. We evaluate four key design choices: (1) **Sample order**: removing this constructs silver traces without ensuring correct samples appear at the end or maintaining a positive ratio of correct to incorrect samples. (2) **Reflections**: removes all `<reflect>` tags and their content from silver traces, concatenating only solution attempts. (3) **Prompt diversity**: Uses only a single prompt ("Let's think step by step") instead of our diverse set P_{solve}. Tests whether varied reasoning patterns matter. Furthermore, we test a variant of the RL-Only method with an **instruction prompt** to encourage `<sample>` and `<reflect>` tag usage through in-context examples, without any SFT stage.

Results on Countdown tasks. All of these methods underperform SkillFactory out-of-domain. Table 5 shows that while RL-Only (Instruction Prompt) performs well on Countdown, it suffers severe degradation on 9 out of 10 OOD tasks, achieving only 25.8% overall accuracy compared to SkillFactory's 32.0%. This pattern holds for both No Sample Order (24.1%) and No Reflections (23.3%), demonstrating that structured SFT traces are essential for cross-domain transfer.

The No Prompt Diversity ablation maintains reasonable performance (29.0% overall) but still underperforms SkillFactory, particularly on computational tasks like Multiplication. This suggests that exposure to diverse reasoning patterns during SFT improves the model's ability to adapt skills to new domains. Additional results suggesting similar findings for Countdown can be found in Appendix B.

These results underscore the importance of key elements of SkillFactory: our use of an explicit SFT stage, the quality of traces we assemble, and the explicit skill demarcation.

7 Conclusion

We introduced SkillFactory, a framework that teaches language models cognitive reasoning skills by restructuring their own outputs into silver traces exhibiting retry and verification patterns. Without requiring stronger teachers, SkillFactory improves performance over baselines on harder task variants as well as across out-of-distribution tasks, and enables inference scaling methods like budget forcing. This self-distillation approach allows us to instill more diverse reasoning skills in language models, making different reasoning capabilities more accessible without distillation.

Reproducibility statement To aid in reproducing SkillFactory, we have given in-depth details about the construction of silver traces in sections 3, including Algorithm 1. Appendices C.2 and C.3 give all of the prompts used in constructing the datasets for training. Additionally, all code, models, and datasets will be made publicly available in future versions of this paper.

REFERENCES

Marah Abdin, Sahaj Agarwal, Ahmed Awadallah, Vidhisha Balachandran, Harkirat Behl, Lingjiao Chen, Gustavo de Rosa, Suriya Gunasekar, Mojan Javaheripi, Neel Joshi, et al. Phi-4-reasoning technical report. *arXiv preprint arXiv:2504.21318*, 2025.

Paul C Bogdan, Uzay Macar, Neel Nanda, and Arthur Conmy. Thought Anchors: Which LLM Reasoning Steps Matter? *arXiv preprint arXiv:2506.19143*, 2025.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training Verifiers to Solve Math Word Problems. *arXiv preprint arXiv:2110.14168*, 2021.

Nouha Dziri, Ximing Lu, Melanie Sclar, Xiang Lorraine Li, Liwei Jiang, Bill Yuchen Lin, Sean Welleck, Peter West, Chandra Bhagavatula, Ronan Le Bras, Jena D. Hwang, Soumya Sanyal, Xiang Ren, Allyson Ettinger, Zaid Harchaoui, and Yejin Choi. Faith and Fate: Limits of Transformers on Compositionality. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=Fkckkr3ya8.

Kanishk Gandhi, Denise H J Lee, Gabriel Grand, Muxin Liu, Winson Cheng, Archit Sharma, and Noah Goodman. Stream of Search (SoS): Learning to Search in Language. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=2cop2jmQVL.

Kanishk Gandhi, Ayush K Chakravarthy, Anikait Singh, Nathan Lile, and Noah Goodman. Cognitive Behaviors that Enable Self-Improving Reasoners, or, Four Habits of Highly Effective STaRs. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=QGJ9ttXLTy.

Arnav Gudibande, Eric Wallace, Charlie Victor Snell, Xinyang Geng, Hao Liu, Pieter Abbeel, Sergey Levine, and Dawn Song. The false promise of imitating proprietary language models. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=Kz3yckpCN5.

Etash Kumar Guha, Ryan Marten, Sedrick Keh, Negin Raoof, Georgios Smyrnis, Hritik Bansal, Marianna Nezhurina, Jean Mercat, Trung Vu, Zayne Sprague, Ashima Suvarna, Benjamin Feuer, Liangyu Chen, Zaid Khan, Eric Frankel, Sachin Grover, Caroline Choi, Niklas Muennighoff, Shiye Su, Wanjia Zhao, John Yang, Shreyas Pimpalgaonkar, Kartik Sharma, Charlie Cheng-Jie Ji, Yichuan Deng, Sarah M. Pratt, Vivek Ramanujan, Jon Saad-Falcon, Jeffrey Li, Achal Dave, Alon Albalak, Kushal Arora, Blake Wulfe, Chinmay Hegde, Greg Durrett, Sewoong Oh, Mohit Bansal, Saadia Gabriel, Aditya Grover, Kai-Wei Chang, Vaishaal Shankar, Aaron Gokaslan, Mike A. Merrill, Tatsunori Hashimoto, Yejin Choi, Jenia Jitsev, Reinhard Heckel, Maheswaran Sathiamoorthy, Alexandros G. Dimakis, and Ludwig Schmidt. OpenThoughts: Data Recipes for Reasoning Models. *CoRR*, abs/2506.04178, June 2025. URL https://doi.org/10.48550/arXiv.2506.04178.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. *arXiv preprint arXiv:2501.12948*, 2025.

Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, et al. OpenAI o1 System Card. *arXiv preprint arXiv:2412.16720*, 2024.

Adam Tauman Kalai, Ofir Nachum, Santosh S Vempala, and Edwin Zhang. Why language models hallucinate. *arXiv preprint arXiv:2509.04664*, 2025.

Joongwon Kim, Anirudh Goyal, Liang Tan, Hannaneh Hajishirzi, Srinivasan Iyer, and Tianlu Wang. ASTRO: Teaching Language Models to Reason by Reflecting and Backtracking In-Context. *arXiv preprint arXiv:2507.00417*, 2025.

Dacheng Li, Shiyi Cao, Tyler Griggs, Shu Liu, Xiangxi Mo, Eric Tang, Sumanth Hegde, Kourosh Hakhamaneshi, Shishir G Patil, Matei Zaharia, et al. LLMs Can Easily Learn to Reason from Demonstrations Structure, not content, is what matters! *arXiv preprint arXiv:2502.07374*, 2025.

Zichen Liu, Changyu Chen, Wenjun Li, Penghui Qi, Tianyu Pang, Chao Du, Wee Sun Lee, and Min Lin. Understanding R1-Zero-Like Training: A Critical Perspective. In *Conference on Language Modeling (COLM)*, 2025.

Sara Vera Marjanović, Arkil Patel, Vaibhav Adlakha, Milad Aghajohari, Parishad BehnamGhader, Mehar Bhatia, Aditi Khandelwal, Austin Kraft, Benno Krojer, Xing Han Lù, et al. DeepSeek-R1 Thoughtology: Let's think about LLM Reasoning. *arXiv preprint arXiv:2504.07128*, 2025.

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. s1: Simple test-time scaling, 2025. URL https://arxiv.org/abs/2501.19393.

Bo Pang, Hanze Dong, Jiacheng Xu, Silvio Savarese, Yingbo Zhou, and Caiming Xiong. BOLT: Bootstrap Long Chain-of-Thought in Language Models without Distillation. *arXiv preprint arXiv:2502.03860*, 2025.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. HybridFlow: A Flexible and Efficient RLHF Framework. *arXiv preprint arXiv: 2409.19256*, 2024.

Parshin Shojaee, Iman Mirzadeh, Keivan Alizadeh, Maxwell Horton, Samy Bengio, and Mehrdad Farajtabar. The Illusion of Thinking: Understanding the Strengths and Limitations of Reasoning Models via the Lens of Problem Complexity. *arXiv preprint arXiv:2506.06941*, 2025.

Joar Skalse, Nikolaus Howe, Dmitrii Krasheninnikov, and David Krueger. Defining and Characterizing Reward Gaming. *Advances in Neural Information Processing Systems*, 35:9460–9471, 2022.

Yang Sui, Yu-Neng Chuang, Guanchu Wang, Jiamu Zhang, Tianyi Zhang, Jiayi Yuan, Hongyi Liu, Andrew Wen, Shaochen Zhong, Na Zou, Hanjie Chen, and Xia Hu. Stop Overthinking: A Survey on Efficient Reasoning for Large Language Models. *Transactions on Machine Learning Research*, 2025. ISSN 2835-8856. URL https://openreview.net/forum?id=HvoG8SxggZ.

Alon Talmor, Jonathan Herzig, Nicholas Lourie, and Jonathan Berant. CommonsenseQA: A question answering challenge targeting commonsense knowledge. In Jill Burstein, Christy Doran, and Thamar Solorio (eds.), *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4149–4158, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1421. URL https://aclanthology.org/N19-1421/.

Qwen Team. Qwen2.5: A party of foundation models, September 2024. URL https://qwenlm.github.io/blog/qwen2.5/.

Yue Wang, Qiuzhi Liu, Jiahao Xu, Tian Liang, Xingyu Chen, Zhiwei He, Linfeng Song, Dian Yu, Juntao Li, Zhuosheng Zhang, et al. Thoughts Are All Over the Place: On the Underthinking of o1-Like LLMs. *arXiv preprint arXiv:2501.18585*, 2025.

Yixin Ye, Zhen Huang, Yang Xiao, Ethan Chern, Shijie Xia, and Pengfei Liu. LIMO: Less is More for Reasoning. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=T2TZ0RY4Zk.

Edward Yeo, Yuxuan Tong, Xinyao Niu, Graham Neubig, and Xiang Yue. Demystifying Long Chain-of-Thought Reasoning in LLMs. In *ICLR 2025 Workshop on Navigating and Addressing Data Problems for Foundation Models*, 2025. URL https://openreview.net/forum?id=AgtQlhMQ0V.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. DAPO: An Open-Source LLM Reinforcement Learning System at Scale. *arXiv preprint arXiv:2503.14476*, 2025.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. STaR: Bootstrapping Reasoning With Reasoning. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (eds.), *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=_3ELRdg2sgI.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. LlamaFactory: Unified Efficient Fine-Tuning of 100+ Language Models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, Bangkok, Thailand, 2024. Association for Computational Linguistics. URL http://arxiv.org/abs/2403.13372.

A Training Hyperparameters

A.1 Hyperparameters: Supervised Fine-tuning

We fine-tune each base model on its own silver traces. We train for one epoch to avoid overfitting. Our goal is not to improve task performance at this stage. Instead, we aim to internalize the cognitive patterns (sampling, reflecting, retrying) that will be refined during RL. We train with a context length of 4096 and use a learning rate of 1e-6 with cosine annealing and full fine-tuning. Training is performed using LlamaFactory (Zheng et al., 2024) with batch size 1.

A.2 Hyperparameters: Reinforcement Learning

We train with RL using GRPO (Shao et al., 2024) on a held-out set of 1,000 questions from the same task distribution, using only binary correctness rewards (1 for correct final answers, 0 for incorrect). This sparse reward signal forces the model to discover which reasoning patterns actually lead to success (Skalse et al., 2022). We train without KL divergence penalties, allowing the model to deviate substantially from its initial policy (Liu et al., 2025; Yu et al., 2025). Our learning rate is 1e-6, batch size 256 with minibatches of 32, and we train for 50 epochs. All experiments are conducted on 4 GH200 GPUs using the VeRL framework (Sheng et al., 2024).

A.3 Generation Parameters: Dataset Construction and Evaluation

We use the standard generation configuration for Qwen2.5-1.5B-Instruct (Team, 2024). More specifically, we use a temperature of 0.7, repetition penalty of 1.1, top_p of 0.8, and top_k of 20.

B Additional Ablation Results

Table 6: Ablation performance on Countdown tasks. Countdown 3arg performance is always high but most ablations underperform SkillFactory on some variant.

Model	**Countdown**			
	3arg	4arg	5arg	6arg
Qwen2.5 1.5B Instruct	14.3 ± 2.1	6.0 ± 1.5	2.0 ± 0.8	0.8 ± 0.5
SkillFactory	98.8 ± 0.5	30.7 ± 2.2	**12.8** ± 1.6	9.0 ± 1.3
Instruction Prompt	97.3 ± 0.7	**32.5** ± 2.5	12.7 ± 1.6	6.7 ± 1.1
No Sample Order	**99.1** ± 0.5	21.8 ± 2.0	10.6 ± 1.4	7.8 ± 1.2
No Reflections	99.0 ± 0.5	20.8 ± 2.1	10.9 ± 1.5	7.5 ± 1.3
No Prompt Diversity	99.0 ± 0.5	31.9 ± 2.2	11.6 ± 1.6	**10.8** ± 1.5
No Reflections	99.0 ± 0.5	20.8 ± 2.1	10.9 ± 1.5	7.5 ± 1.3

Table 6 shows that all methods achieve near-perfect accuracy on the training task (Countdown 3-arg), indicating that design choices primarily affect generalization rather than in-distribution performance. On harder variants, two patterns emerge. First, removing reflections or proper sample ordering significantly hurts performance (e.g., 20.8% vs. 30.7% on 4-arg for No Reflections). This suggests that both explicit verification and solution quality are crucial for learning transferable skills. Second, the RL-Only (Instruction Prompt) baseline achieves competitive performance on Countdown variants (32.5% on 4-arg), even surpassing SkillFactory slightly.

C DATA CURATION

C.1 GLUE PHRASES

Glue phrases are phrases that are placed between the <sample> <reflect> tags. These serve to guide the model to generate a new solution. We categorize our glue phrases into three types: phrases for correct responses, phrases for incorrect responses, and generic glue phrases. The phrases for correct responses reaffirm that the previous answer was correct, but still prompt the model to give a new response. For instance, *"This previous answer was correct, but I should double check it to be sure."* Meanwhile, the phrases for incorrect responses verbalize that the previous answer was incorrect and that the model should generate a new reasoning trace. An example is *"My previous answer was incorrect. I will now try again."* Lastly, generic glue phrases are neutral and do not depend on whether the previous answer was correct or incorrect. An example is *"But wait, let me think about it again."*

While constructing the SkillFactory SFT dataset, we add a glue phrase after every sample-reflection sequence. If the sample-reflection sequence yielded a correct answer, we sample from correct_glue_phrases ∪ generic_glue_phrases. If the sample-reflection sequence yielded an incorrect answer, we sample from incorrect_glue_phrases ∪ generic_glue_phrases. The set of glue phrases were first generated by an LLM from a few hand-written seed prompts, then manually filtered and edited for clarity and diversity. The complete set of glue phrases is listed below:

- generic_glue_phrases = [‘‘However, I should double check this answer.", ‘‘But wait, let me think about it again.’’, ‘‘I can resolve this question to be sure.’’, ‘‘Let me verify my answer.’’, ‘‘I should check my response again.’’, ‘‘I can double check my response.’’, ‘‘Wait...’’, ‘‘Wait! I should double check my answer.’’, ‘‘Although, if I want to be absolutely sure, I should do this again.’’, ‘‘I’ll recheck what I said earlier.’’, ‘‘Time to review my response one more time.’’]
- correct_glue_phrases = [‘‘This previous answer was correct, but I should double check it to be sure.’’, ‘‘Let me try this question again to verify that my response is actually correct.’’, ‘‘My earlier answer seems correct, but I should double check it to be sure.’’, ‘‘That response looks right, and I have verified it. It might be worth doing it again just in case.’’ ‘‘That answer seems fine, but I’d like to double check for to be safe.’’, ‘‘I believe that was the right answer, but let me make sure.’’, ‘‘My previous response looks accurate, though I should recheck it.’’, ‘‘The solution seems right. I will now retry it to be more confident.’’, ‘‘Looking back, my earlier answer seems right, though I’ll recheck it.’’ ‘‘I’m fairly confident the last answer was right, but I’ll double-check anyway.’’ ‘‘That response looks solid, though I want to be certain.’’, ‘‘I’m leaning toward my last answer being right, but I’ll test it once more.’’ ‘‘It’s better to be cautious | I’ll re-verify my previous answer.’’, ‘‘Seems right to me, but a second look won’t hurt.’’]
- incorrect_glue_phrases = [‘‘My previous answer was incorrect. I will now try again.’’, ‘‘On review, my last response falls short, so I’ll attempt a new one.’’ ‘‘After reconsideration, I can see my earlier answer wasn’t right, and I’ll try again.’’, ‘‘I learned from my mistake in the last answer | let me rework it.’’, ‘‘I may have missed the mark earlier. Let me rethink and attempt again.’’, ‘‘Instead of sticking with my incorrect answer, I’ll try a new approach.’’, ‘‘Oops, I see the issue now | time for another try.’’, ‘‘I realize that wasn’t the right answer. Let’s fix it.’’, ‘‘I see the flaw in my earlier response. I’ll try a new one.’’, ‘‘I made an error before, so I’ll reconsider and answer again.’’, ‘‘Oops, that wasn’t right. Let me take another shot.’’, ‘‘Looks like I messed up earlier. I’ll go again.’’, ‘‘Since my earlier answer was incorrect, I’ll rework the reasoning and attempt again.’’, ‘‘My last attempt wasn’t correct, but I’ll refine it and try again.’’]

Table 7: Values for the parameters used in Algorithm 1

Parameter	Value
D_T	Countdown 3-Arg
N_{sample}	16
L_{max}	5

C.2 Prompt Variants

We use the following prompt variants

1. **Original**: "Let's think step by step."
2. **Plan and execute**: "To solve this question, write a high level plan you intend to use starting with "First, I'll try to understand the problem better by writing out a plan and go really deep into detail about how I should solve this," then execute that plan (whatever reasoning is required), then give your resulting {answer_type_str} as the answer in the "<answer>(your answer)</answer>" tag."
   - System prompt: "You like to solve problems by understanding the problem, writing a plan, executing the plan, then giving an answer. Write a plan that when reasoned over would solve the question then give your answer in <answer>(your answer)</answer>. You always end with </answer>, you never ever end without giving an answer."
3. **Alternatively**: "Think step by step and find some potential answers using the word "Alternatively," to distinguish them when you are discussing if they are correct, then give your resulting {answer_type_str} as the answer in the "<answer>(your answer)</answer>" tags."
   - System prompt: "You like to find multiple answers for a question then deliberate over them saying "Alternatively," between each answer you are deliberating on and then you give your final answer in "<answer>(your answer)</answer>". You always end with </answer>, you never ever end without giving an answer."
4. **Rephrase**: "Begin your response with "Rewritten Question: " and by rewriting the question making it contain only what is needed to solve it, then think step by step and then give your resulting {answer_type_str} as the answer in the "<answer>(your answer)</answer>" tags."
   - System prompt: You answer questions by saying "Rewritten Question: " then rewriting the question to only contain what is needed to solve it and then think step by step and then you give your final answer in "<answer>(your answer)</answer>". You always end with </answer>, you never ever end without giving an answer."

C.3 Reflection Prompts

We use the following prompts to prompt the model to generate reflections:

Reflection Prompt for Acronym task

```
Below is a question and a model response.
After reading the question and the model response, please reflect on whether the
model response is correct or incorrect.
Do not attempt to correct the model response or to improve it, just reflect on it.

# Problem
{x['question']}

# Model Response
{x[response_col][0]}
```

```
# Task
Is this previous answer correct or incorrect? Reflect on it and add your final
answer inside <verdict> </verdict> tags.

To give another example, if the list of words was [ "iota", "disrespecting",
"essentials", "mashup", "analyse" ] and the target is to come up with at least
four letter valid english word, and the answer the model response gives you was
'ema', you could write:
Let us verify this answer: 'ema'. First, let me check if the response uses the
first letters of the given word in order: the first letters of each word in the
given list are: 'i', 'd', 'e', 'm', 'a'. The letters in the given answer are:'e',
'm', 'a'. Yes the responses uses the first letter of the words in order.
Then, let me check if the response is at least four letters long, no it is not.
Then, let me check if the response is an english word, no it is not.
Since the response violates constraints in the prompt, it is incorrect.
<verdict>
Incorrect
</verdict>

To give another example,  if the list of words was [ "iota", "disrespecting",
"essentials", "mashup", "analyse" ] and the target is to come up with at least
four letter valid english word, and the answer the model response gives you was
'idea', you could write:
Let us verify this answer: 'idea'. First, let me check if the response uses the
first letters of the given word in order: the first letters of each word in the
given list are: 'i', 'd', 'e', 'm', 'a'. The letters in the given answer are: 'i',
'd', 'e', 'a'. Yes the responses uses the first letter of the words in order.
Then, let me check if the response is at least four letters long, yes it is.
Then, let me check if the response is an english word, yes it is.
Since the response satisfies all constraints in the prompt, it is correct.
<verdict>
Correct
</verdict>

Remember, only reflect on the model response, do not attempt to correct it or
improve it.
Report your final assessment inside <verdict> </verdict> tags. You may only say a
verdict is "Correct" or "Incorrect". Nothing else is allowed within the <verdict>
tags. Make your reflections brief, but you should always reflect before the
<verdict> tags, you cannot only give a verdict. Start your response with "Let us
verify this answer:". Do not answer the question, determine if the models answer
is correct.
```

Reflection Prompt for the Letter Countdown task

```
Below is a question and a model response.
After reading the question and the model response, please reflect on whether the
model response is correct or incorrect.
Do not attempt to correct the model response or to improve it, just reflect on it.

# Problem
{x['question']}

# Model Response
{x[response_col][0]}

# Task
Is this previous answer correct or incorrect? Reflect on it and add your final
answer inside <verdict> </verdict> tags.
```

```
To give another example, if the list of letters was ['f','t','s','r','e','a'] and
the target is to come up with at least four letter valid english word using
letters from the input, and the answer the model response gives you was 'trace',
you could write:
Let us verify this answer: 'trace'. First, let me check if the response uses
letters from the input: 't' is in the input, 'r' is in the input, 'a' is in the
input, 'c' is not in the input, 'e' is in the input. The answer uses a letter not
in the input list.
Then, let me check if the response is at least four letters long, yes it is since
the answer is 5 letters long, which is greater than 4.
Then, let me check if the response is an english word, yes it is.
Since the response violates constraints in the prompt, it is incorrect.
<verdict>
Incorrect
</verdict>

To give another example, if the list of letters was ['f','t','s','r','e','a'] and
the target is to come up with at least four letter valid english word using
letters from the input, and the answer the model response gives you was 'fast',
you could write:
Let us verify this answer: 'fast'. First, let me check if the response uses
letters from the input: 'f' is in the input, 'a' is in the input, 's' is in the
input, 't' is in the input. The answer uses letters from the input list.
Then, let me check if the response is at least four letters long, yes it is since
the answer is 4 letters long.
Then, let me check if the response is an english word, yes it is.
Since the response satisfies all constraints, it is correct.
<verdict>
Correct
</verdict>

Remember, only reflect on the model response, do not attempt to correct it or
improve it.
Report your final assessment inside <verdict> </verdict> tags. You may only say a
verdict is "Correct" or "Incorrect". Nothing else is allowed within the <verdict>
tags. Make your reflections brief, but you should always reflect before the
<verdict> tags, you cannot only give a verdict. Start your response with "Let us
verify this answer:". Do not answer the question, determine if the models answer
is correct.
```

Reflection Prompt for the GSM8k task

```
Below is a question and a model response.
After reading the question and the model response, please reflect on whether the
model response is correct or incorrect.
Do not attempt to correct the model response or to improve it, just reflect on it.

# Problem
{x['question']}

# Model Response
{x[response_col][0]}

# Task
Is this previous answer correct or incorrect? Reflect on it and add your final
answer inside <verdict> </verdict> tags.
```

```
For example, if the question was "Marc bought 5 model cars that cost $20 each and
5 bottles of paint that cost $10 each. He also bought 5 paintbrushes that cost $2
each. How much did Marc spend in total?" with the models response answering "5 x
20 = 100. 5 x 10 = 50. 5 x 2 = 10. 100 + 50 = 150. The answer is 150." you could
write:
Let us verify this answer: The model breaks the question down into subparts. 5 x
20 is 100. 5 x 10 is 50. 5 x 2 is 10. But then it only adds 100 + 50 and doesn't
add the 10 to the final answer. Therefore this is likely incorrect since we want
the absolute total.
<verdict>
Incorrect
</verdict>

To give another example, if the question was "Crackers contain 15 calories each
and cookies contain 50 calories each. If Jimmy eats 7 cookies, how many crackers
does he need to eat to have consumed a total of 500 calories?" with the models
response answering "7 x 50 = 350. 500 - 350 = 150. 150 / 15 = 10. 10 is the
answer.", you could write:
Let us verify this answer: To answer this question, we need to know how many
calories Jimmy ate, subtract that from 500, then divide it by the average calories
in a cracker. The model does this exactly. First finding 7 x 50 = 350 which is
correct. Then it subtracts this from 500 getting 150, again, correct. Finally, it
takes the remaining 150 calories and divides it by 15 to get 10. This is most
likely correct.
<verdict>
Correct
</verdict>

Remember, only reflect on the model response, do not attempt to correct it or
improve it.
Report your final assessment inside <verdict> </verdict> tags. You may only say a
verdict is "Correct" or "Incorrect". Nothing else is allowed within the <verdict>
tags. Make your reflections brief, but you should always reflect before the
<verdict> tags, you cannot only give a verdict. Start your response with "Let us
verify this answer:". Do not answer the question, determine if the models answer
is correct.
```

Reflection Prompt for the CSQA task

```
Below is a question and a model response.
After reading the question and the model response, please reflect on whether the
model response is correct or incorrect.
Do not attempt to correct the model response or to improve it, just reflect on it.

# Problem
{x['question']}

# Model Response
{x[response_col][0]}

# Task
Is this previous answer correct or incorrect? Reflect on it and add your final
answer inside <verdict> </verdict> tags.

For example, if the question was "What establishment uses a revolving door as a
security measure?" with the answer choices being "A: a bank" and "B: Gamestop",
with the models response answering "Games are valuable and Gamestop is a place of
business which needs security, therefore, Gamestop is the answer." you could
write:
```

```
Let us verify this answer: Gamestop probably does not have revolving doors nor is
in need of security despite it being a place of business, this is because a bank
seems much more likely to need security, therefore I think the given answer is
incorrect.
<verdict>
Incorrect
</verdict>

To give another example, if the question was "What home entertainment equipment
requires cable?" with the answer choices being "A: a sink", "B: a bed", and "C: a
television" with the models response answering "A television requires cable and is
most likely the right answer here.", you could write:
Let us verify this answer: A sink doesn't really require electricity except for
the garbage disposal, a bed (with the exception of a few special types of beds)
also does not use electricity. A TV however, always needs a cable and electricity
to run. Additionally people also say "do you have cable" referring to a type of
service for the television. Overall, the model ignored explaining away the other
answers, but correctly identified the answer that most likely is correct therefore
I believe the models answer is correct..
<verdict>
Correct
</verdict>

Remember, only reflect on the model response, do not attempt to correct it or
improve it.
Report your final assessment inside <verdict> </verdict> tags. You may only say a
verdict is "Correct" or "Incorrect". Nothing else is allowed within the <verdict>
tags. Make your reflections brief, but you should always reflect before the
<verdict> tags, you cannot only give a verdict. Start your response with "Let us
verify this answer:". Do not answer the question, determine if the models answer
is correct.
```

Reflection Prompt for the Long Multiplication task

```
Below is a question and a model response.
After reading the question and the model response, please reflect on whether the
model response is correct or incorrect.
Do not attempt to correct the model response or to improve it, just reflect on it.

# Problem
{x['question']}

# Model Response
{x[response_col][0]}

# Task
Is this previous answer correct or incorrect? Reflect on it and add your final
answer inside <verdict> </verdict> tags.

For example, if the question was "100 x 100" with the models response answering
"100 x 100 = 100 x 10 + 100 x 10 = 1000 + 1000 = 2000" you could write:
Let us verify this answer: The reasoning is trying to breakdown the arithmetic
into two subproblems that are easier to solve. This is good. But the subproblems
are wrong. You cannot add two 100 x 10 together to get 100 x 100. Therefore this
is incorrect.
<verdict>
Incorrect
</verdict>

To give another example, if the question was "200 x 350" with the models response
answering "2 x 35 = 70. 70 x 100 = 7,000. 7,000 x 10 = 70,000. The answer is
70,000.", you could write:
```

```
Let us verify this answer: The model broke the multiplication down into steps.
First it multiplies 2 x 35, ignoring the 0s, to make the problem easier. 2 x 35 is
indeed 70. Then it starts to multiply the result, 70, with the magnitudes of each
operand (100 for the first operand and 10 for the second). This results in 70,000
which seems correct.
<verdict>
Correct
</verdict>

Remember, only reflect on the model response, do not attempt to correct it or
improve it.
Report your final assessment inside <verdict> </verdict> tags. You may only say a
verdict is "Correct" or "Incorrect". Nothing else is allowed within the <verdict>
tags. Make your reflections brief, but you should always reflect before the
<verdict> tags, you cannot only give a verdict. Start your response with "Let us
verify this answer:". Do not answer the question, determine if the models answer
is correct.
```

Reflection Prompt for the Countdown task

```
Below is a question and a model response.
After reading the question and the model response, please reflect on whether the
model response is correct or incorrect.
Do not attempt to correct the model response or to improve it, just reflect on it.

# Problem
{x['question']}

# Model Response
{x[response_col][0]}

# Task
Is this previous answer correct or incorrect? Reflect on it and add your final
answer inside <verdict> </verdict> tags.

For example, if the list of numbers was [20, 28, 98], the target was 658, and the
answer was 98 + 28 \\times 20 = 658 you could write:
Let us verify this answer: First, 28 \\times 20 = 560, then 98 + 560 = 658. The
answer is correct.
<verdict>
Correct
</verdict>

To give another example, if the list of numbers was [22, 8, 38], the target was
214, and the answer was (22 - 8) \\times 38 = 214, you could write:
Let us verify this answer: First, 22 - 8 = 14, then 14 \\times 38 = 532. This is
not equal to 214, so the answer is incorrect.
<verdict>
Incorrect
</verdict>

Remember, only reflect on the model response, do not attempt to correct it or
improve it.
Report your final assessment inside <verdict> </verdict> tags. You may only say a
verdict is "Correct" or "Incorrect". Nothing else is allowed within the <verdict>
tags. Make your reflections brief, but you should always reflect before the
<verdict> tags, you cannot only give a verdict. Start your response with "Let us
verify this answer:". Do not answer the question, determine if the models answer
is correct.
```

D MODEL OUTPUTS

In this section, we show examples from the RL baseline and the SkillFactory model.

For in domain variants of tasks like Countdown 4 argument, the RL only model does implicit retrying and verification. However, responses from our SkillFactory model have implicit and explicit reflections.

RL Only output for a Countdown 4arg Task

```
To create an equation using the numbers 60, 20, 1, and 11 that equals 880,
we can follow these steps:

1. First, add 60 and 20:
   Step 1: 60 + 20 = 80
2. Next, add the result to 1:
   Step 2: 80 + 1 = 81
3. Finally, add the result to 11:
   Step 3: 81 + 11 = 92

However, 92 is not equal to 880. Let's try a different approach:

1. First, multiply 60 and 11:
   Step 1: 60 * 11 = 660
2. Then, add 20 to the result:
   Step 2: 660 + 20 = 680

Again, 680 is not equal to 880. Let's try a different combination:

1. First, multiply 20 and 11:
   Step 1: 20 * 11 = 220
2. Then, add 60 to the result:
   Step 2: 220 + 60 = 280

This is also not equal to 880. Let's try a different set of operations:

1. First, multiply 60 and 1:
   Step 1: 60 * 1 = 60
2. Then, add 20 to the result:
   Step 2: 60 + 20 = 80

Finally, add the result to 11:
   Step 3: 80 + 11 = 91

This is also not equal to 880. Let's try a different approach:

1. First, multiply 20 and 11:
   Step 1: 20 * 11 = 220
2. Then, subtract 60 from the result:
   Step 2: 220 - 60 = 160

This is also not equal to 880. Let's try a different set of operations:

1. First, multiply 60 and 1:
   Step 1: 60 * 1 = 60
2. Then, add 20 to the result:
   Step 2: 60 + 20 = 80
Then, multiply the result by 11:
   Step 3: 80 * 11 = 880
```

```
This is equal to 880. So, the equation is:

<answer>
(60 * 1) * 11
</answer>
```

SkillFactory models output for a Countdown 4arg task

```
<think>
<sample>
To solve for 880 using the numbers 60, 20, 1, and 11 with each number used
exactly once, we need to find a combination of addition, subtraction,
multiplication, and division that will yield 880. Let's break down the
process:

1. **Identify possible combinations**: We start by considering different
ways to combine these numbers.
2. **Trial and error**: Since there are limited combinations, we will try
various permutations to see if any work.

Let's start with some trial:

- **Combination 1**: Multiply 60 and 11 first, then add 20 and 1.
  \[
  (60 * 11) + 20 + 1 = 660 + 20 + 1 = 681
  \]
  This does not equal 880.

- **Combination 2**: Multiply 20 and 11 first, then add 60 and 1.
  \[
  (20 * 11) + 60 + 1 = 220 + 60 + 1 = 281
  \]
  This also does not equal 880.

- **Combination 3**: Subtract 1 from 60, then multiply the result by 20,
and finally add 11.
  \[
  (60 - 1) * 20 + 11 = 59 * 20 + 11 = 1180 + 11 = 1191
  \]
  This exceeds 880.

- **Combination 4**: Add 60 and 20, then multiply the sum by 11.
  \[
  (60 + 20) * 11 = 80 * 11 = 880
  \]
  This works!

Therefore, the correct combination is:
\[
(60 + 20) * 11 = 880
\]

So, the final answer is:
<answer>
(60 + 20) * 11
</answer>
```

```
</sample>
<reflect>
Let us verify this answer: First, 60 + 20 = 80, then 80 * 11 = 880. The
answer is correct.
</reflect>
</think>
That confirms the final answer is:
<answer>
(60 + 20) * 11
</answer>
```

RL Only output for a Long Multiplication 3 digit Task

```
To solve the multiplication equation \(520 \times 133\), we can break it
down into simpler steps:

First, let's multiply 520 by 3:
\[520 \times 3 = 1560\]

Next, let's multiply the result by 100 (since 133 is 100 + 33):
\[1560 \times 100 = 156000\]

Now, let's multiply 520 by 13:
\[520 \times 13 = 6760\]

Finally, add the two results together:
\[156000 + 6760 = 162760\]

So, the solution to the multiplication equation \(520 \times 133\) is
162760.

Therefore, the final answer is:

<answer>
162760
</answer>
```

SkillFactory models output for a Long Multiplication 3 digit task

```
<think>
<sample>
To solve the multiplication equation \( 520 \times 133 \), we will break
it down into simpler steps:

### Step 1: Break down 133 into more manageable parts.
\[ 133 = 100 + 30 + 3 \]

### Step 2: Multiply 520 by each part of 133 separately.

#### Part 1: \( 520 \times 100 \)
\[ 520 \times 100 = 52000 \]

#### Part 2: \( 520 \times 30 \)
\[ 520 \times 30 = 15600 \]

#### Part 3: \( 520 \times 3 \)
\[ 520 \times 3 = 1560 \]

### Step 3: Add the results from each part together.
\[ 52000 + 15600 + 1560 \]

### Step 4: Perform the addition.
\[ 52000 + 15600 = 67600 \]
\[ 67600 + 1560 = 69160 \]

Therefore, the product of \( 520 \times 133 \) is **69160**.

The final answer is:
<answer>
69160
</answer>
</sample>
<reflect>
Let us verify this answer:
\[ 520 \times 133 = 520 \times (100 + 30 + 3) = 520 \times 100 + 520
\times 30 + 520 \times 3 = 52000 + 15600 + 1560 = 69160 \]
The calculation is correct.
</reflect>
</think>


That confirms the final answer is:

<answer>
69160
</answer>
```

RL Only output for GSM8k Task

```
To solve this problem, we need to follow these steps:

1. Determine the speed of the car in the slow lane.
2. Calculate the time it takes for the car in the slow lane to travel 480
miles.

Step 1: Determine the speed of the car in the slow lane.
The car in the fast lane travels at 60 miles per hour. The car in the slow
lane travels at half that speed:
\[ \text{Speed of slow lane car} = \frac{60}{2} = 30 \text{ miles per
hour} \]

Step 2: Calculate the time it takes for the car in the slow lane to travel
480 miles.
We use the formula:
\[ \text{Time} = \frac{\text{Distance}}{\text{Speed}} \]
Substitute the known values:
\[ \text{Time} = \frac{480 \text{ miles}}{30 \text{ miles per hour}} = 16
\text{ hours} \]

So, the car in the slow lane took 16 hours to cover the same distance of
480 miles.

Therefore, the final answer is:
<answer>
(16)
</answer>
```

SkillFactory models output for a GSM8k

```
<think>
<sample>
To solve this problem, we need to follow these steps:

1. **Determine the speed of the slower car**:
   - The faster car travels at 60 miles per hour.
   - The slower car travels at half the speed of the faster car.
   \[
   \text{Speed of slower car} = \frac{60}{2} = 30 \text{ miles per hour}
   \]

2. **Calculate the time taken by the faster car**:
   - Distance traveled by the faster car = 480 miles.
   - Speed of the faster car = 60 miles per hour.
   \[
   \text{Time taken by faster car} = \frac{\text{Distance}}{\text{Speed}}
   = \frac{480 \text{ miles}}{60 \text{ miles per hour}} = 8 \text{ hours}
   \]

3. **Calculate the time taken by the slower car**:
   - Distance traveled by the slower car = Same as the faster car (since
   both travel the same distance).
   - Speed of the slower car = 30 miles per hour.
   \[
   \text{Time taken by slower car} = \frac{\text{Distance}}{\text{Speed}}
   = \frac{480 \text{ miles}}{30 \text{ miles per hour}} = 16 \text{
   hours}
   \]

Therefore, the time taken by the slower car to cover the same distance is
16 hours.

The final answer is:
<answer>
16
</answer>
</sample>
<reflect>
Let us verify this answer: First, we calculate the speed of the slower
car, which is \( \frac{60}{2} = 30 \) mph. Then, we use the formula \(
\text{time} = \frac{\text{distance}}{\text{speed}} \) to find the time
taken by the slower car: \( \frac{480}{30} = 16 \) hours. This confirms
our answer is correct.
</reflect>
</think>


That verifies the final answer is:

<answer>
16
</answer>
```

RL Only output for a Letter Countdown 4 Task

```
To solve this problem, we need to find the longest possible English word
that can be formed using only the given letters. Let's analyze the input
letters: ['p', 'a', 's', 's'].

1. First, let's count how many times each letter appears in the input:
   - p appears 1 time
   - a appears 1 time
   - s appears 2 times

2. Since we cannot form more than two 's' letters with just these four
letters, the longest word we can form will have at most three 's' letters
and some other letters.

3. Let's try forming words with different combinations of the letters:
   - "sssa": This gives us the word "ssssa".
   - "ssas": This gives us the word "ssass".

4. Both "ssssa" and "ssass" are longer than any single letter
combination, but we need to check if they can actually be formed from the
given letters.

5. "ssssa":
   - We have 1 's' left.
   - We have 1 'a'.
   - We don't have enough 'p' or 's'.

6. "ssass":
   - We have 1 's', 1 'a', and 2 's'.
   - This gives us the word "ssass".

Since "ssass" is a valid English word and it uses all the letters
provided, we can conclude that the longest valid word is "ssass".

Therefore, the answer is: <answer> ssass </answer>
```

SkillFactory models output for a Letter Countdown 4 task

```
<think>
<sample>
To solve this problem, we need to follow these steps:

1. **Identify Available Letters**: First, we identify all the unique
letters available from the given set.
   - Input letters: `['p', 'a', 's', 's']`

2. **Check Valid Words**: Next, we check which combinations of these
letters form valid English words that meet the criteria (at least 4
characters long).

3. **Formulate Word**: We then select the longest valid word from our
checks.

Let's start with identifying the unique letters:
- Unique letters: {'p', 'a', 's'}

Now, let’s try forming words using these letters:
- Possible words starting with 'p':
  - "pass"

- Possible words starting with 'a':
  - "asps"

- Possible words starting with 's':
  - "spas"
  - "asp"

Among these, the longest valid word is "pass".

Therefore, the longest valid word formed from the given letters is "pass".
</sample>
<reflect>
Let us verify this answer: The word "pass" uses all the letters 'p', 'a',
's', and 's' exactly once. It meets the condition of being at least 4
characters long and consists only of the provided letters. Therefore, the
answer is correct.
</reflect>
</think>


That confirms the final answer is:

<answer>pass</answer>
```

In example outputs below we see differences in how the two models generalize the skill use. In the RL only output, we see the model pre-supposing a candidate answer to be an English word and then checking for other constraints. There is also no explicit reflection of the final answer or implicit or explicit retrying. Compared to the RL only output, our proposed method generates to explicit answer attempts followed by explicit reflections. We also see differences in reasoning strategies where our proposed model retries with different strategies (enumerating constraints and high level criteria vs trying step by step).

RL Only output for Acronym 4 letter word Task

```
To construct an acronym that meets all the given conditions, let's
analyze the list of words: ['relax', 'evolution', 'aesthetic', 'line'].

1. **Condition 1**: The acronym must be at least 4 characters long.
   - The words have lengths: 5, 9, 8, and 4 respectively.
   - The shortest word has 4 characters, so we need to include the first 4
   words in our acronym.

2. **Condition 2**: The acronym must use the first letter of at least 4
words in the list.
   - The first letters of the words are: r, e, a, l.
   - We can choose any 4 out of these 4 letters. Let's pick r, e, a, l.

3. **Condition 3**: The acronym should only use letters from a subset of
the words in the list, in order.
   - We will use the first 4 words: relax, evolution, aesthetic, line.

4. **Condition 4**: The acronym should produce a valid English word.
   - Let's combine the chosen letters: realevate.

Now, let's check if "realevate" is a valid English word:
- It contains all the required letters from the selected words.
- It follows the order of the selected words.
- It forms a meaningful word.

Therefore, the acronym that satisfies all the conditions is: <answer>
realvate </answer>
```

SkillFactory output for a Acronym 4 letter word Task

```
<think>
<sample>
To solve this problem, we need to follow these steps:
1. **Identify Key Criteria**:  - The acronym must be at least 4
characters long. - It should use the first letter of at least 4 words from
the list.   - Only letters from specific words should be used.   - The
resulting acronym must form a valid English word.
[...truncated due to space...]
Therefore, my final answer is:
<answer> REAL </answer>
</sample>
<reflect>
[..natural language reasoning..]
<verdict>
Incorrect
</verdict>
</reflect>
I should double check my response.
<sample>
To construct an acronym that meets the given criteria, let's break down
the process:
### Step-by-Step Solution
#### Condition 1: At Least 4 Characters
We need an acronym with at least 4 characters.
#### Condition 2: Uses the first letter of at least 4 words
From the list ["relax", "evolution", "aesthetic", "line"], we'll take the
first letters of each word:
- relax -> r
- evolution -> e
- aesthetic -> a
- line -> l
This gives us the initial set {r, e, a, l}. To ensure we have at least 4
unique letters, we can add another distinct letter from the remaining
words. Let's choose "e" again for consistency but consider other
possibilities later.
#### Condition 3: Only uses letters from a subset of the words
Since we already used "e" twice, let's try adding another distinct letter
from the available options. We can pick "v" from "evolution".
So now our set becomes {r, e, v, a, l}.
#### Condition 4: Produces a valid English word
Next, we need to combine these letters into one word that forms a real
English word. One possible combination is "real".
Thus, the final acronym that meets all the criteria is:

<answer>real</answer>
</sample>
<reflect>
Let us verify this answer: The acronym formed using the first letters of
"relax", "evolution", "aesthetic", and "line" is indeed "real". This
satisfies all the given conditions: 1. It has exactly 4 characters. 2. It
uses the first letter of "relax", "evolution", "aesthetic", and "line".
3. All letters come from the specified words. 4. It forms the valid
English word "real".
Therefore, the answer is correct.
</reflect>
</think>

That confirms the final answer is:
<answer>real</answer>
```

Table 8: Number of explicit answer attempts, explicit reflections and the verification F1 for the correct and incorrect classes for Skill Factory.

	#Answers	#Reflect	F1 (correct / incorrect)
Acronym 4	2.84	2.32	0.180 / 0.840
Acronym 5	2.76	2.36	0.140 / 0.860
CSQA	3.40	1.78	0.210 / 0.640
Countdown 3arg	1.57	1.22	0.960 / 0.910
Countdown 4arg	1.21	3.06	0.680 / 0.950
Countdown 5arg	0.98	3.03	0.660 / 0.980
Countdown 6arg	0.99	2.99	0.700 / 0.980
GSM8k	1.99	1.98	0.490 / 0.760
Letter Countdown 4	2.12	1.65	0.340 / 0.790
Letter Countdown 5	2.07	1.69	0.180 / 0.790
Long Multiplication 2dig	2.21	1.29	0.530 / 0.360
Long Multiplication 3dig	2.15	1.84	0.360 / 0.800
Long Multiplication 4dig	2.46	2.20	0.110 / 0.860
Long Multiplication 5dig	2.39	2.00	0.010 / 0.850

D.1 Analysis of Skill Use

We report skill use by SkillFactory across all tasks in Table 8.

E Additional Details for Our BOLT-like Baseline

We randomly sample 10 questions from our training split of **Countdown with 3 arguments** and prompt `claude-sonnet-4-20250514` to produce high-quality reasoning traces for each question with the following user prompt.

Prompt for High Quality Reasoning Traces from Claude Sonnet 4

```
{x['question]'}

Your response must not only solve the problem but also deliberately include the
following elements: a clear problem analysis, an explicit plan, exploration of
alternative solution paths, explicit backtracking when a path fails, reflection
on your choices, verification of both intermediate steps and the final result, and
strict adherence to the required output format. Including these components is just
as important as arriving at the correct answer.
```

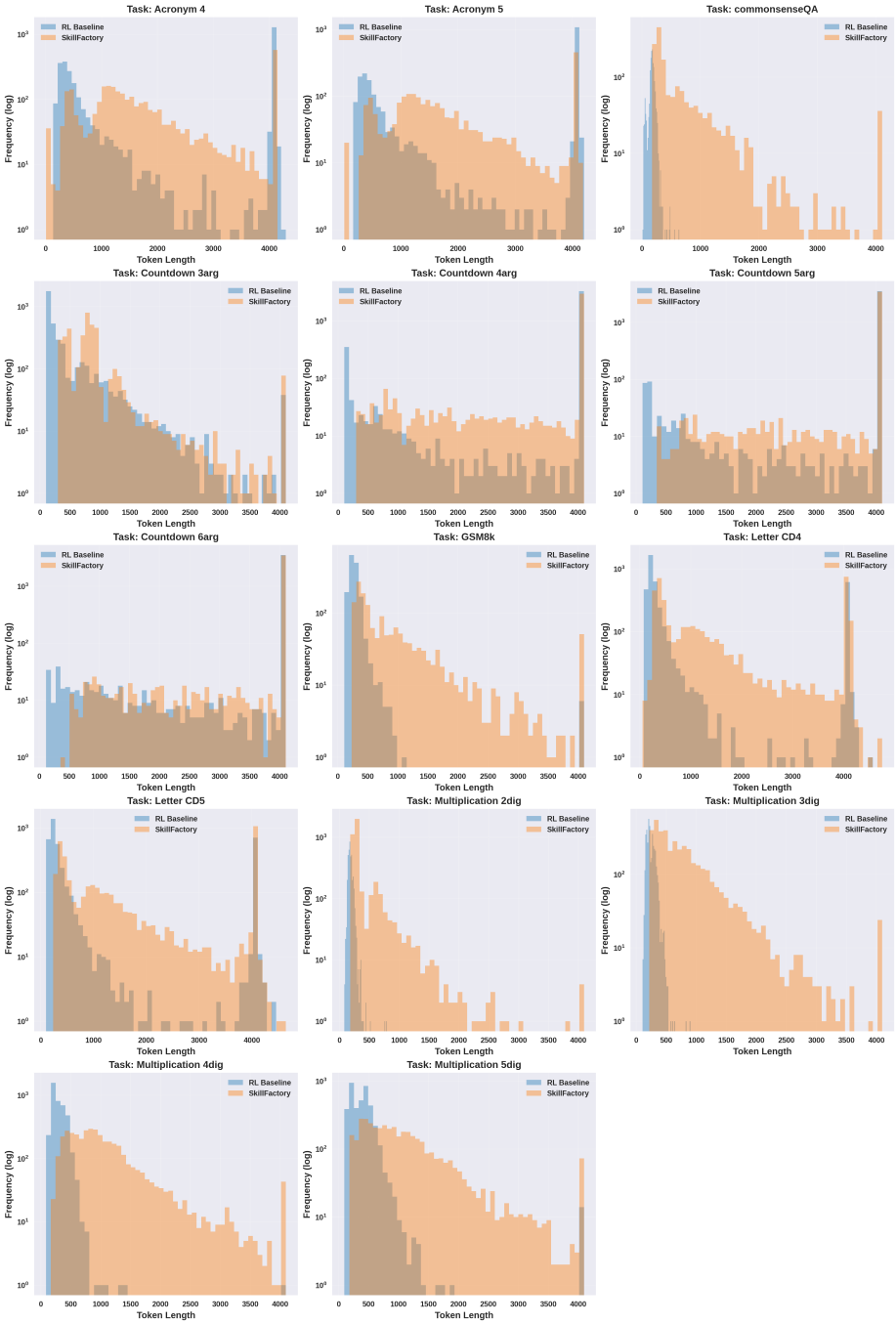

Figure 5: Distribution of token response of all responses given by two models: RL Baseline and SkillFactory (proposed method).

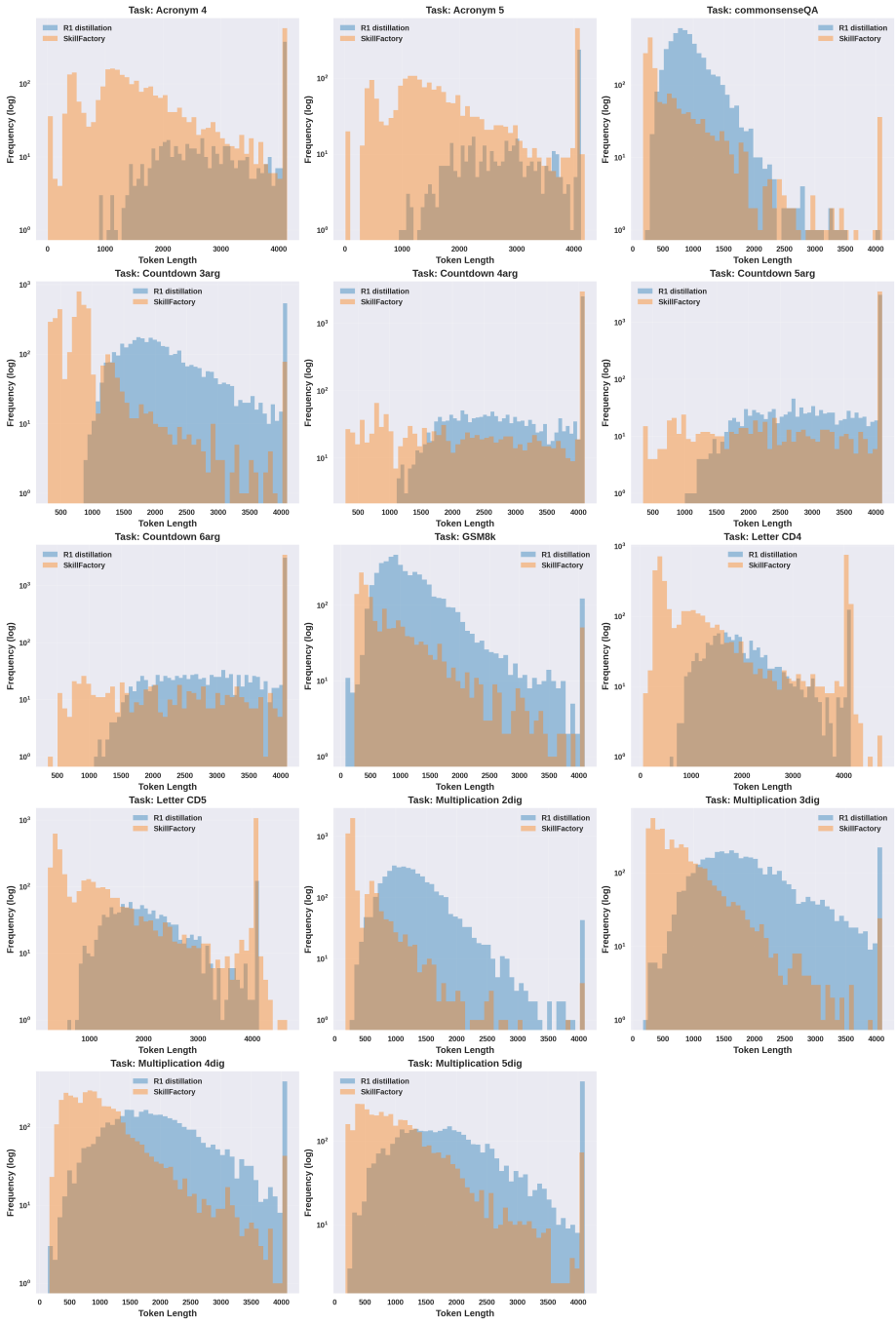

Figure 6: Distribution of token response of all responses given by two models: R1 Distillation and SkillFactory (proposed method).

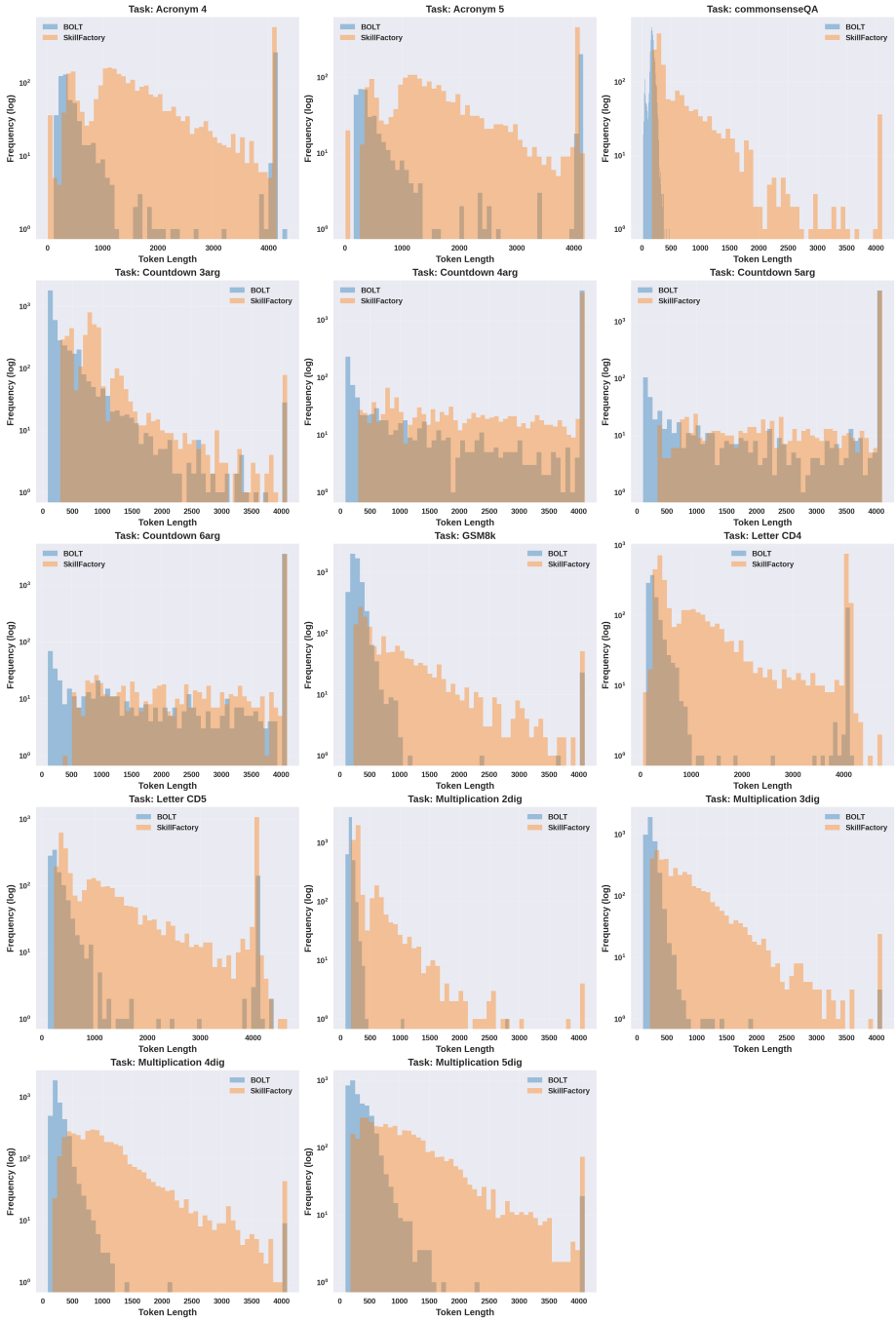

Figure 7: Distribution of token response of all responses given by two models: BOLT and SkillFactory (proposed method).

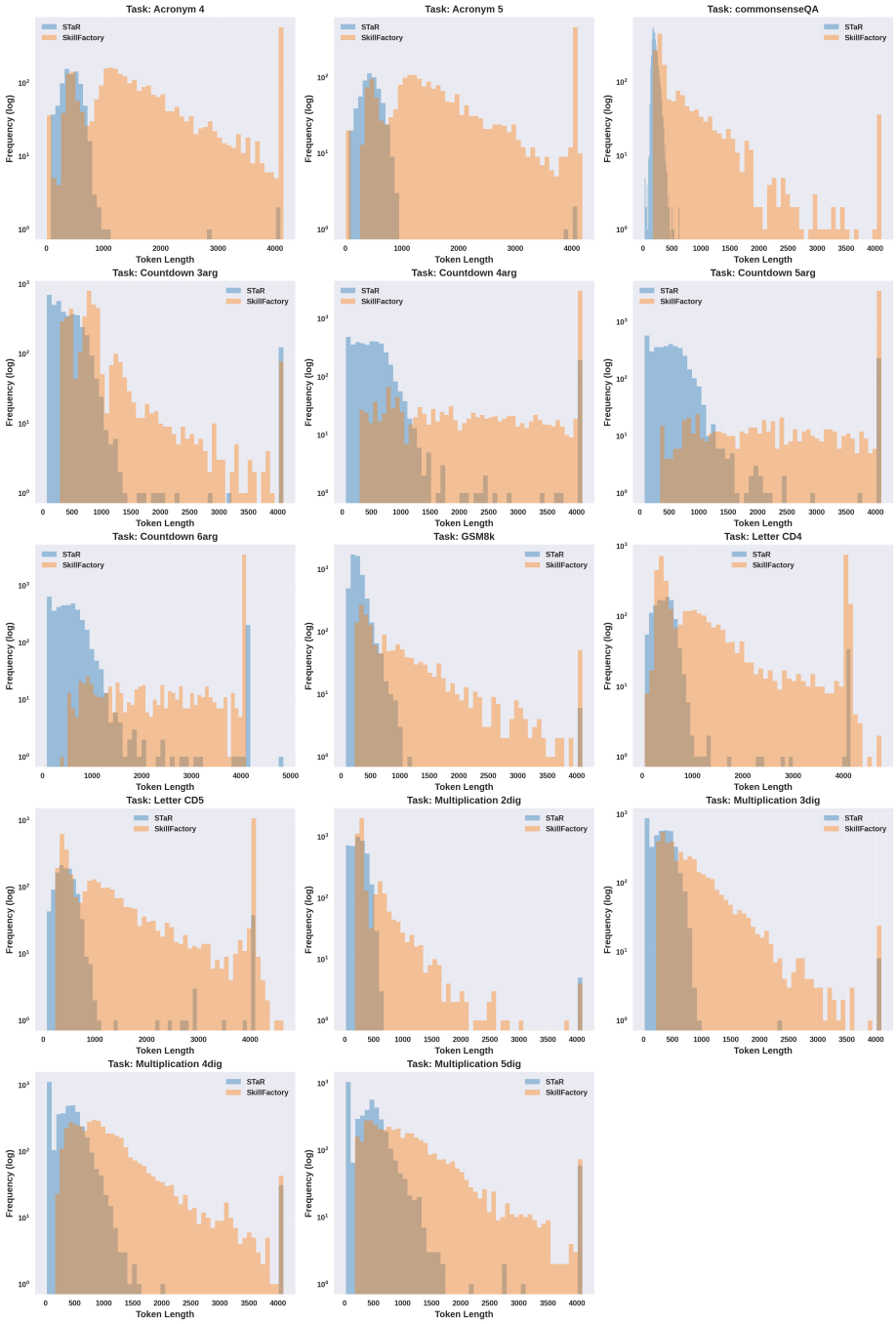

Figure 8: Distribution of token response of all responses given by two models: STaR and SkillFactory (proposed method).

F LLM CONTRIBUTIONS

We used LLMs mainly to help with minor tweaking of Latex formats and to jumpstart ideas for figure one that were ultimately heavily edited and expanded upon. Additionally, minor usage to reword paragraphs for conciseness that were then edited and rephrased by the authors heavily (if not rewritten entirely).